

MIDTERM EXAM

- This test contains 12 questions with a total of 75 points
 - 1-9 5 points each
 - 10-12 10 points each
- You have 50 minutes to complete this midterm.
- You may **not** use your text, or any other reference material.
- Do not turn this page or turn over the exam until instructed to do so.

Hints

- Don't spend too much time on short answer questions. If you don't know the answer right away, move on and come back.
- There may be some questions that will require more time to think and organize your thoughts. If you find yourself stuck on one of these questions, save it until you have answered all the questions you can answer quickly. If you spend too much time on one question you may get frustrated and that will impair your ability to answer the other questions.
- Don't turn your test in early. If you have time left over, use it to review your answers. Students who turn tests in early often make trivial mistakes that they would catch if they reviewed their answers.
- Students that get the highest grades usually go over their answers several times.
- The space below a question **does not** indicate how long of an answer I am expecting. Sometimes I want to start the next question on a new page.
- Double check that you understand the question before you answer it. If you don't understand the question, ask me.

1) (5 points) Describe how an operating system starts running when the machine is turned on.

A program called a boot loader is executed when the machine turns on. Usually a very simple boot loader is stored in non-volatile memory. This simple boot loader loads a more complex boot loader from a well known location on a hard drive. The complex boot loader loads the operating system.

2) (5 points) What advantages do threads have over processes?

threads share memory making it easier to program
responsiveness – faster context switching and faster to create

3) (5 points) What is a program counter? Describe (in detail) how it is used.

A program counter holds the address in memory of the next instruction to be executed for this process. The CPU uses the program counter to determine which instruction to load next. The instruction immediately after the current instruction can be executed by incrementing the program counter. When a program contains a jump (if statement, function call, etc) the program counter is set to the first instruction in the target block.

The program counter is crucial during a context switch. It keeps track of where the process is to resume execution the next time it is running.

4) (5 points) What is the run time stack? Include an explanation of when items are added and removed from the stack. Giving an example and drawing a picture may help explain it.

When a function is called, information about the function is placed into a structure called an activation record. The run time stack is a stack of activation records. When a function is called its activation record is placed on the run time stack. When the function terminates its activation record is popped from the run time stack.

5) (5 points) What would cause a process to switch states from:

running --> blocked/waiting

Process requests a resource or performs I/O. While the process is waiting for the resource it is placed on the blocked/waiting queue.

ready --> running

The scheduler picks the process to be the next one to execute. The dispatcher switches the context to this process.

blocked/waiting --> running

When the resource the process was waiting for became available or the I/O the process was waiting for completed, the process's state would be switched to ready. Then the process would be picked to be the next running process (see above).

6) (5 points) What happens during a context switch? Include a description of the mode (kernel/user).

The execution mode is switched from user to kernel. All information about the currently executing process is saved to the process' PCB. The PCB is placed on the ready queue. The scheduler picks the next process to execute. This process' PCB is removed from the ready queue. All the information about the new process is loaded into CPU registers. The CPU's program counter is restored. The mode is switched from kernel to user. The process is started.

7) (5 points) What is the difference between a preemptive scheduler and a non-preemptive scheduler? Describe how preemption is initiated.

A non-preemptive scheduler lets processes run until they terminate or block.

A preemptive scheduler sets a maximum time (called a time quantum) that each process can execute. If a process is still executing when the maximum time has passed, it loses its turn using the CPU – it is preempted. The process is placed back on the ready queue.

This is achieved by setting a countdown timer before starting each process. The timer sends an interrupt to the CPU after the predetermined amount of time. When the CPU handles the interrupt it preempts the process.

8) (5 points) What are the ramifications of preemptive scheduling at the kernel level? At the user level?

User: program can get interrupted at any time. Must write the code so there are no race conditions.

Kernel level: processes could get interrupted when in the middle of a kernel call. This could result in data inconsistencies.

9) (5 points) What factors should be considered when determining the length of a time quantum (time slice)?

Average CPU burst length: want to pick a time quantum long enough so most processes block or terminate before using up their time quantum

Responsiveness: want a time quantum short enough so the system provides adequate responsiveness.

Context switching time: if the time quantum is too short, too much time will be spent switching contexts. In other words, the CPU will spend so much time switching contexts that significantly less real work will be done.

10) (10 points) On multiprocessor machines each processor usually has its own scheduler. These schedulers could share a single ready queue or each of them could maintain a separate ready queue. What are the advantages and disadvantages of all the schedulers sharing a ready queue? What are the advantages/disadvantages of each scheduler having its own queue?

Shared queue: (+) perfect load balancing (-) processes are often moved between processors, because processors have multiple sets of registers a process can be switched out of the CPU and still have its register values stored in the CPU, so it is faster for a process to keep using the same CPU (processor affinity) (-) several CPUs may become idle while waiting to access the shared ready queue.

Queue for each processor: (+) quick scheduling (+) can take advantage of processor affinity (-) no easy means for load balancing.

11)(10 points) From a process scheduling perspective, outline what happens when a program reads a number from a file and writes it to the screen. Your outline should include the disk, memory, CPU, and interrupts.

process is running

process requests to read the first number

CPU sends request to disk-driver

process is placed on a blocked queue

CPU does something else

disk-driver reads the number from the disk and puts it directly into main memory

I/O completes and the disk-driver interrupts the CPU

the process is moved to the ready queue

at some point the process gets to run

process retrieves the number from main memory

process writes the number to the screen by calling a function in the windowing system

12) (10 points) The following is a working program: it executes without error. What does it print? No credit unless you explain your answer. This question is more difficult than it appears.

```
int
main()
{
    pid_t child1 = fork();
    assert(child1 >= 0);

    if (child1 == 0)
    {
        cout << "child1\n";
    }
    pid_t child2 = fork();
    assert(child2 >= 0);

    if (child2 == 0)
    {
        cout << "child2\n";
    }

    while (wait(NULL) > 0)
        ;
    cout << "parent\n";
}
```

The problem with this program is that both child processes fail to exit after executing the code in their if-pid-equals-zero-blocks. This means that child1 forks a child2 process, calls wait(), and prints parent. Child2 calls wait() (which does nothing) and then prints parent. The following is printed. The order may be different depending on how the scheduler schedules the processes.

```
child1
child2
parent
parent
child2
parent
parent
```